Speedy Sudoku Solver

Caden Brown, Gavin Roake, Caleb Freckmann

***Abstract*—Sudoku is a popular number-puzzle game that is found in newspapers or magazines. This project is to explore the parallelization of traditionally sequential Sudoku solvers. The most conventional algorithm used to solve sudoko is backtracking via DFS, which is effective but, is computationally expensive and difficult to parallelize due to its stack-based nature. To overcome this, we experiment with a hybrid approach, implementing human method-inspired algorithms, as elimination, lone ranger, twins, and triplets—for preprocessing, followed by a multithreaded brute-force phase that divides the search space across threads. We also examine early exit parallelization to short-circuit redundant constraint checks, as well as Dancing Links. While several methods showed promise, especially for processing multiple puzzles in parallel, many thread-level optimizations incurred overhead exceeding their benefit. Ultimately, the most practical and performant solution was found to be a hybrid model using sequential logic to reduce complexity before applying parallel brute-force techniques.**

I Problem Statement

The goal of this project aims to parallelize the sequential solutions that solve these Sudoku puzzles. Familiar topics such as multithreading, conditional variables, and mutexes will be used to solve this problem.

II Backtracking-Based Solver

Firstly, the most popular algorithm when it comes to solving Sudoku boards involves backtracking using a Depth-First Search on the grid.

The structure of Sudoku lends itself well to a solution thats makes use of backtracking. However backtracking tends to be computationally expensive. Although backtracking tends to be computationally expensive, the use of multi-threading will allow the Sudoku solver to be optimized through multiple sub-grids. Additionally, synchronization, for shared results, and locking, to prevent race conditions, of the threads needs to occur in areas where there are shared rows or columns. Incorporating multi-threading can significantly enhance performance through leveraging multiprocessing and exploring multiple possibilities simultaneously.

II-A Basic Anatomy of the Backtracking Algorithm

1) **Base Case:** Check if the grid is full.
2) (Optional) Check for duplicate states.
3) Generate possible moves:
   (a) **Check validity of the move:** Ensure each number does not violate Sudoku rules.
   (b) **Change state:** Place the number on the board.
   (c) **Perform recursive descent:** Solve the next possible cell in the grid.
   (d) **(Optional) Process return value of the recursive call:** Was it a success? If so, propagate the result (i.e., return).
   (e) **Undo state change:** Remove the number to backtrack.
4) As a result of running our algorithm, it will return the completed Sudoku or an error message if the board is unsolvable.

II-B Why Parallel DFS Fails

Since backtracking is a DFS, it is not directly parallelizable due to the stack. Threads cannot work all on the same stack causing early exits

```
if (in final cell of board)
        return true
if (in last column)
        go to next row, col = 0
if (board[row][col] != 0)
        return solveSudokuSequentialBacktracking(board, row, column)

make state change:
for (i in 1 - 9 possible numbers)
    if (isSafeMove)
      board[row][column] = i;
      if (solveSequentialBacktracking)
          return true;
      undo state change: board[row][column] = 0

return false
```

Fig. 1. PseudoCode Example

or higher contention than the sequential algorithm.

III Human-Like Strategies for Solving Sudoku

Another algorithm for solving Sudoku leverages typical strategies for solving Sudoku combined with backtracking to optimize execution time. The standard approaches that will be implemented are elimination, lone ranger, twins, and triplets.

Elimination is the most fundamental Sudoku-solving strategy. It is based on the fact that each cell in a row, column, or 3×3 box can contain only one unique number from 1 to 9. If a number is already placed in a row, column, or box, it can be eliminated as a possible candidate for other empty cells in that region. The process involves scanning all known numbers and updating the possible values for remaining empty cells accordingly. If a cell ends up with only one valid number left, that number is placed in the cell. This process is repeated iteratively since new eliminations may occur once new numbers are placed. While simple, this strategy is highly effective in the early stages of solving, as it can rapidly reduce the complexity of the puzzle by restricting the possibilities.

The Lone Ranger strategy identifies numbers that appear as a candidate in only one cell within a given row, column, or box. Even if other numbers remain as possibilities in that cell, the fact that this number cannot go anywhere else forces it to be assigned there. For example, if a row has five empty cells, and the number 7 can only fit into one of them (even if that cell also has other candidates), then 7 must be placed there. This strategy is particularly useful when elimination alone is not enough to determine placements. Once a Lone Ranger is found and placed, elimination can further simplify the puzzle, creating a cascading effect that makes solving easier.

The Twins strategy, also known as Naked Pairs, applies when exactly two cells in the same row, column, or box contain the same two possible numbers and no others. These numbers form a locked pair, meaning that they must go into those two cells. As a result, no other cell in that row, column, or box can contain those numbers. This allows for the elimination of those numbers from all other candidates in the affected area. For instance, if two cells in a row have only 5 and 7 as possible values, then other cells in that row cannot have 5 or 7. This strategy significantly reduces the number of possibilities and is an essential tool for progressing through difficult puzzles.

The Triplets strategy, also called Naked Triples, works similarly to Twins but involves three numbers distributed across three cells. If exactly three cells in a row, column, or box contain the same three numbers as their only possible candidates, then those numbers must be placed within those three cells. Any other cell in the same row, column, or box must exclude these three numbers from its possibilities. For example, if three cells in a box have only 1, 2, and 4 as candidates, then other cells in that box cannot contain 1, 2, or 4. The Triplets strategy is particularly useful in tougher puzzles where possibilities remain widespread, and applying it can cause a major breakthrough in solving the puzzle efficiently.

These strategies work together dynamically, as finding a solution using one method can

create new opportunities for the others. The algorithm cycles through them iteratively until no further progress can be made, at which point a brute-force approach is used if necessary. The brute-force method uses depth-first search (DFS) but optimizes performance by pre-filling the first seven empty cells with valid permutations. These partial boards are pushed onto a shared stack and then processed in parallel by multiple threads. Each thread pops a partial board and attempts to complete it using DFS. The first thread to find a valid solution stops all others. To avoid race conditions, a lock mechanism is implemented to ensure that no two threads modify overlapping regions simultaneously. Since the algorithm runs in parallel, multiple threads may try to modify overlapping cells simultaneously, leading to race conditions.

The primary issue occurs when:

- Two threads attempt to update the same cell simultaneously – If one thread assigns a value while another is also computing a valid number, inconsistencies arise.
- A thread modifies a row, column, or box while another thread is using it for validation – This could result in incorrect eliminations or invalid assumptions about possible numbers.
- Threads compete for stack access – If too many small workloads are assigned, threads may frequently compete to pop from the stack, leading to inefficiencies.

To mitigate these issues, the algorithm employs locking mechanisms:

- Locking a conflict boundary: When updating a cell, the entire row, column, and box are temporarily locked to prevent simultaneous modifications.
- Non-blocking work reassignment: If a thread encounters a locked cell, it selects a different work item rather than waiting, reducing idle time and improving efficiency.

By implementing these strategies, the algorithm ensures a robust and efficient parallel Sudoku solver that balances speed with correctness. When parallelizing the strategies previously mentioned, their runtime ended up being much slower than iteratively. For example, the parallel elimination strategy took 1.32059 ms to execute while the standard iterative approach took 0.039394 ms. This is the case for the other approaches as well. Parallelism pays off when each iteration is computationally heavy. But in elimination, you're just checking a set, setting a number, and erasing a few values. The required thread creation, context switching, and locks introduce way more cost than the few microseconds saved. Since a sudoku board is quite small, any updates cause a lot of collisions. Due to the fact, most threads will spend more time waiting than working. So it makes no sense to execute these in parallel, instead these approaches will be used to preprocess the board to make it go faster.

IV Multithreaded Brute Force

An iterative brute force solution attempts every possible combination of digits in every single possible cell until the correct one is found.

For each empty cell on the Sudoku board, we may have up to n valid digits to try. Thus, in the worst case, the total number of possibilities can be as high as:

$$n^{(number of empty cells)}$$

With 10 empty cells, there are about 3.5 billion combinations, and this could take a time of 1 hour to finish.

To solve this, we introduce the aspect of multiple threads working in chunks during the Brute Force Process. We calculate the total number of valid number combinations for the empty cells. Then, the program divides the search for solutions into chunks based on n number of available threads. Once a solution is found, update the solution found flag.

```
const unsigned long long total_combinations = static_cast<unsigned long long>(pow(SIZE, emptyCells.size()));
const int num_threads = omp_get_max_threads();
const unsigned long long chunk_size = max(1ULL, total_combinations / num_threads);

#pragma omp parallel
{
    int thread_num = omp_get_thread_num();
    unsigned long long start = thread_num * chunk_size;
    unsigned long long end = (thread_num == num_threads - 1) ? total_combinations : min(start + chunk_size, total_combinations);

    solveChunk(board, emptyCells, start, end);
}

if (solution_found)
{
    board = solution;
    return true;
}
return false;
```

Fig. 2. Parallel Brute Force Code

V Early Exit Parallelization

Now, imagine you and your two friends working together to determine if a number can be placed at a certain position on the board without conflicts. The thing about you and your friends is that each of you can only check for one type of conflict. That is, one of you can only see the column, one can see just the row, and the last can only see the grid.

As of right now, a number is placed, then it is first checked by one person for row accuracy, then the second person checks for column accuracy, and finally, the last person checks for the sub-grid accuracy. If the number cannot be placed in the sub-grid, both the row checker and the column checker wasted precious time checking. This is not optimal, as it is sequential where it does not need to be.

But alas, there is a solution.

What if all three could look and assess the board at the same time in such a way that if even one of the three checkers runs into a problem, they can all stop immediately? This, for example, would save the effort of checking whether a number is okay in the row when it won't be okay in the column or sub-grid. This is a form of early exit parallelization. The time saved would be the difference between a longer check and an early exit.

Each checker will have to perform a maximum of 8 checks when running concurrently. This is a speed-up when compared to the maximum of 20 (not 24, since running sub-grid last will have a guaranteed 4 cells checked already in row/col verification).

One thing worth noting is that there is a small overhead added because before each cell can be checked, the checker must first see if any of the other threads "shout to stop." This is a minimal amount of overhead, as boolean checking in an optimized language such as C++ adds very little clock time in comparison to the complex cell check performed regardless. When decompiled, this extra logical check adds a maximum of 3 lightweight atomic ops (a move, a compare, and a jump). This is an insignificant overhead when compared to the heavy cell check (vector-of-vector memory access with comparison), which the preliminary check seeks to avoid (see clock comparison results for estimated speed-up).

The statistical analysis on how many checks this saves depends on the algorithm used to generate guesses; furthermore, it is a complicated analysis that will only yield an estimate of the expected performance gain under certain assumptions about guess quality, failure distribution, and board configuration. This is why we will lean on empirical performance analysis instead, to estimate the speed-up factor.

V-A Overlapping Cell Checks

If for each proposed placement we check the 8 row cells, 8 column cells, and 8 sub-grid

cells, this will ensure a 4-cell overlap between checks. This is not very optimal and seems like a no-brainer to improve. While eliminating redundant checks appears efficient on the surface, it risks compromising the structure that enables us to short-circuit on earlier collisions.

But if we remove this doubling of effort, we also prevent an early exit with the proposed optimization in certain cases. For example, if the sub-grid has to go through 8 checks to find the collision, whereas the row check only has to go through 2 checks, this is a waste of 6 checks. This illustrates a potential downside: we might be forced to scan a longer sequence just to detect the same issue.

What matters is how often this occurs. And with a little bit of napkin math, it is concluded that this case is uncommon enough to ignore the 4-check improvement on all bad case improvements. The 3x8 = 24 checks include 4 overlapping ones, so optimizing would save 4 checks per placement. But this prevents early exits — e.g., if a row check would find a conflict in 2 steps, but we now wait for the sub-grid to take 6, we've lost time. Unless that happens in more than 10% of cases, the tradeoff isn't worth it.

VI Imitation of Parallelism: A Sequential Alternative

VI-A Sequential Parallelism Simulation

After further testing, it is apparent that the overhead associated with thread allocation is far too high for such a small and frequent operation. If not for the overhead of thread allocation and joining, then this would be a viable strategy.

This leaves us with the next best option: imitation. By sequentially "imitating" the parallel solution, we can still make a set of the total number of cases run in a more average time. We can think of this solution as showing the three checkers with one number all at once. The program will check if the first cell of the row is an issue, then the first cell of the column, and then the first cell of the sub-grid. This way, a collision in the sub-grid is called after the third iteration, not the 19th.

While not as fast as the theoretical parallel solution, it certainly offers some actual gain.

VII Dancing Links

Dancing Links is an efficient algorithm developed by Donald Knuth to solve exact cover problems. In the context of Sudoku, the rows hold the potential placements of a number while the columns represent a constraint that must be satisfied. In this case, no repeat numbers in the rows, columns, and subgrid. The key part of this algorithm is the data structure powering it, circular doubly linked lists which enable rapid removal and reinsertion of rows and columns, a crucial step in the recursive backtracking search.

Throughout solving, the Dancing Links algorithm selects the column (constraint) with the fewest options to minimize branching. Following that, it then tries each possible number placement in each row, covering all associated constraints, and recursively proceeds. When it hits a dead end, it backtracks by uncovering the columns. The cover/uncover mechanism allows Dancing Links to efficiently explore the solution space with minimal overhead. The combination of these steps allows it to be extremely fast for solving sudoku puzzles and similar problems containing a large number of constraints.

Parallelizing the Dancing Links algorithm is challenging and often counterproductive due to its tightly coupled nature and pointer heavy structure. The core of the algorithm relies on recursively selecting columns, covering/uncovering nodes, and backtracking. All of which are operations that involve frequent and precise manipulation of a shared data structure. Parallel execution in this context requires significant locking or deep copying of the linked structure, which introduces substantial

overhead and complexity. When implementing on our own, we found that the algorithm is so optimized that the overhead from parallelizing actually was slower than our sequential implementation.

VIII Multiple Puzzle Parallelization

Even though most sudoku solving algorithms are difficult to parallelize for a single board, solving multiple puzzles in parallel is a highly effective and efficient approach as compared to solving the puzzles one by one. Each puzzle is an independent task with no shared state, making them ideal candidates for parallel execution. By assigning a separate thread or task to each puzzle, systems can fully utilize multicore processors without the overhead of synchronization or shared memory management that would otherwise slow down concurrent solutions on the same board. This implementation scales well, when dealing with large batches of puzzles. Workloads are evenly distributed across all puzzles and are significantly reduced, making it valuable for applications like benchmarking solvers, generating puzzle datasets, or running competitions and test suites.

In practice, implementing this involves running each puzzle through the algorithm that the user decides. This strategy ensures optimal CPU utilization and simplifies the architecture compared to trying to parallelize individual solving algorithms, which tend to involve complex recursive logic or tightly-coupled data structures. Multiple puzzle parallelization is therefore a practical and scalable solution when speed and efficiency are critical.

IX Conclusion

Throughout the project, we explored various algorithms for solving Sudoku through parallel techniques. The structure of Sudoku lends itself well to a solution which makes use of backtracking, and although this tends to be computationally expensive, the use of multithreading allowed for performance improvements through parallel exploration. However, parallel DFS fails due to high contention on a shared stack, and human-like strategies such as Elimination or Lone Ranger, while effective sequentially, introduce more overhead than benefit when parallelized.

By implementing locking mechanisms and early exit strategies, we sought to address race conditions and reduce unnecessary computation, but observed that parallelism pays off only when each iteration is computationally heavy. For lightweight operations like Elimination, parallelization actually slows performance. As a result, these strategies are best used to preprocess the board, rather than being executed in parallel.

Dancing Links proved to be extremely fast in its sequential form, but its pointer-heavy structure made parallelization inefficient and counterproductive. On the other hand, multiple puzzle parallelization stood out as an excellent use of parallelism, as puzzles are independent and do not require shared memory. This method scales well and fully utilizes system resources without introducing significant overhead.

Overall, a thorough search of potential parallel solutions to Sudoku was explored as intended. Unfortunately, as with many research projects, this search was mostly fruitless. However, we did learn a lot in the process, and in the end, the hybrid approach—preprocessing with logical strategies followed by multithreaded brute force—offered the best balance between speed and correctness.